Static Multiple-Issue Processors: VLIW Approach

Instructor: Prof. Cristina Silvano, email: cristina.silvano@polimi.it
Teaching Assistant: Dr. Giovanni Agosta, email: agosta@acm.org

Dipartimento di Elettronica, Informazione e Bioengegneria
Politecnico di Milano

Nov 2013
Summary

- Recap on Instruction Level Parallelism
- VLIW Architectures
- Code Scheduling for VLIW Architectures
  - Next lecture
Instruction Level Parallelism
Recap on Single-Issue Processors

- Until now:
  - RISC architecture
  - MIPS Instruction Pipeline
  - Pipeline Hazards (Instruction Control/Data Dependencies)
  - Branch prediction mechanisms (Dynamic - Static)

- Techniques/Optimizations to exploit ILP on Single Issue Architectures
  - **Ideal CPI = 1** → 1 instruction takes 1 cycle to execute, in best case
Recap on Multiple Issue Processors

 IPC_{\text{ideal}} = 2; CPI_{\text{ideal}} = 0.5
Towards Multiple-Issue Processors

- **Single-Issue Processors**: Scalar processors that fetch and issue max one operation in each clock cycle.

- **Multiple-Issue Processors** require:
  - Fetching more instructions in a cycle (higher bandwidth from the instruction cache)

- **Multiple-Issue Processors** can be:
  - **Dynamically scheduled** (issue a varying number of instructions at each clock cycle).
  - **Statically scheduled** (issue a fixed number of instructions at each clock cycle).
Recap on Dynamically Scheduled Processors

- Number of issued instructions per cycle varies (1-8)
  - 2+ → Superscalar Processors; CPI = 1/issued instructions
- Scheduling can be performed purely in hardware
  - See Tomasulo’s algorithm
  - However, the compiler can improve the quality of the schedule
- Superscalar Processors are common in general purpose (desktop/server) computing
  - Most Intel processors from Pentium on
  - Alpha, modern PowePC, Sparc, etc.
- However, hardware techniques such as Tomasulo are costly:
  - PowerPC 750 Instruction Sequencer ~=70% ALU+FPU+LSU
  - Cycle time limited by delay of scheduling logic
  - Design verification becomes expensive
Recap on Dynamic Scheduling

- The **hardware** rearranges the instruction execution to reduce pipeline stalls while maintaining data flow and exception behavior.

- **Main advantages:**
  - It enables handling some cases where dependences are unknown at compile time
  - It simplifies the compiler complexity
  - It allows compiled code to run efficiently on a different pipeline.

- Those advantages are gained at a cost of a significant increase in hardware complexity and power consumption.
Compilers can use sophisticated algorithms for code scheduling to exploit ILP (Instruction Level Parallelism)

- Detect whether two instructions can be parallelized
- Schedule them so they will be executed in parallel

Problem: the amount of parallelism available within a basic block is small (<3 in generic code)

- Basic Block: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
- Example: For typical MIPS programs the average branch frequency is between 15% and 25% ⇒ from 4 to 7 instructions execute between a pair of branches
Data dependences can further limit the amount of ILP we can exploit within a basic block to much less than the average basic block size

- True data dependences force sequential execution of instructions
- The compiler can, however, deal to some extent with false data dependences

To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks (i.e. across branches).
Determining dependences among instructions is critical to defining the amount of parallelism existing in a program.

If two instructions are dependent, they cannot execute in parallel: they must be executed in order or only partially overlapped.

Three different types of dependences:
- **Data Dependences** (or True Data Dependences)
- **Name Dependences**
  - Anti-Dependence: WAR
  - Output Dependence: WAW
- **Control Dependences**
Program Properties

- **Two properties** are critical to program correctness (and normally preserved by maintaining both data and control dependences):

  - **Data flow**: Actual flow of data values among instructions that produces the correct results and consumes them.

  - **Exception behavior**: Preserving exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program.
VLIW Processors:
An Alternative Way of Extracting ILP
VLIW Processors

- Problem: Hardware-based dependence checking and dynamic scheduling for superscalar processors are area and power consuming.
- Basic idea: to reduce the amount of area and power consumed by making the compiler to statically choose what operations can be executed in parallel.
- These parallel ops are packaged by the compiler into a single issue packet (bundle) so that hardware needs not checking explicitly for dependences.
- The compiler must ensure that dependences within the issue packet cannot be present (pure VLIW) or, at most, indicate when a dependence may be present (Explicitly Parallel Instruction Computer, EPIC).

(Statically Scheduled) Very Long Instruction Word Processor (VLIW)
VLIW Processors

- VLIW approach advantages:
  - Simpler hardware
  - Low power consumption
  - Good performance through extensive compiler optimization

- The single issue packet (bundle) represents a wide instruction (64, 128 or more bits) with multiple operations per instruction
  - Thus the name "Very Long Instruction Word"

- Early VLIWs were quite rigid in the instruction format and they required recompilation of programs for different versions of the hardware
VLIW processors

- The long instruction (bundle) has a set of fields for each Functional Unit (called slots): for example 2, 4, 5, or more slots.
- Example: a 5-issue VLIW has a long instruction that can contain up to 5 operations (corresponding to 5 slots) including 1 integer operation (or a branch), 2 floating point ops and 2 memory references.
- Decode unit is reduced to simple decode logic (for each op)
- It is eventually present a dispatch network that redirects ops and related operands to FUs.
- To keep the FU busy, there must be enough parallelism in the code sequence to fill the available operation slots.
Pipelined VLIW Architecture Overview

- IF
- ID
- RR
- EX
- WB
VLIW Processors: Operation Latency

- VLIW instructions are not atomic, i.e., latency of operations is exposed in the source code:

  \[ I \quad [C=A*B, \ldots]; \]
  \[ I+1 \quad [NOP, \ldots]; \text{compiler inserted} \]
  \[ I+2 \quad [X=C*F, \ldots]; \]

- In this case "*" has a latency=2
  - The compiler must schedule the use of C after 1 instruction
  - Otherwise correct execution is compromised
  - True even in the case of a pipelined multiplier
VLIW Processors: Dependences

- True, anti and output dependencies are solved by the compiler and not by the hardware by taking into account FU latency.

- **RAW Hazards:**
  - Scalars/superscalars generate NOPs/stalls or execute successive instructions (dynamic scheduling or instruction reordering).
  - In VLIW other instructions are statically inserted by the compiler during the scheduling phase.
    - Ideally, instructions not involved in the dependencies should be used.
    - Otherwise, the compiler can generate **NOPs**.
VLIW Processors: Dependences

- **WAR** and **WAW** hazards are generally solved by the compiler by correctly selecting temporal slots for the operations or by register renaming.

- **Structural hazards** are also solved by the compiler.

- Compiler can provide useful information on how to statically predict **branches**.

- **Control hazards** are solved by the hardware by flushing execution of incorrectly predicted branches.
**VLIW code bundles: A simple example**

<table>
<thead>
<tr>
<th>INSTRUCTIONS</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
<th>C9</th>
<th>C10</th>
<th>C11</th>
<th>C12</th>
<th>C13</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1: lw$2, BASEA($4)</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $2,$2, INC1</td>
<td></td>
<td>X</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $3, BASEB($4)</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $3,$3, INC2</td>
<td></td>
<td>X</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $4, $4, 4</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne $4,$7, L1</td>
<td></td>
<td>X</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS schedule**
**VLIW code bundles: A simple example**

<table>
<thead>
<tr>
<th>INSTRUCTIONS</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
<th>C9</th>
<th>C10</th>
<th>C11</th>
<th>C12</th>
<th>C13</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1: lw$2,BASEA($4)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $2,$2,INC1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $3,BASEB($4)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $3,$3,INC2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $4, $4, 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne $4,$7, L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MIPS schedule**

**4-issue VLIW:**
- 2 LD/ST, 1 Int FU, 1 Int/Br FU

<table>
<thead>
<tr>
<th>SLOT1: LD/ST Ops</th>
<th>SLOT2: LD/ST Ops</th>
<th>SLOT3: Integer Ops</th>
<th>SLOT4: Integer+Branch Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw$2,BASEA($4)</td>
<td>lw $3,BASEB($4)</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>addi $2,$2,INC1</td>
<td>addi $3,$3,INC2</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>addi $4, $4, 4</td>
<td>NOP</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>bne $4,$7, L1</td>
</tr>
</tbody>
</table>
Advantages of VLIW

- Compiler can use sophisticated algorithms to schedule code (exploiting program parallelism) to increase performance
  - The compiler can observe the code on a much wider scale than the hardware → think the instruction window observed by the Tomasulo hardware!
- Instructions have fixed fields ⇒ easier to decode
- Reduced hardware complexity
  - Small die area
  - Low Processor Cost
  - Low Power Consumption
  - Easily extended to larger number of FUs
Challenges of VLIW Technology

- Need for strong Compiler Technology
  - Detect and exploit parallelism
  - Need to manage parallelism beyond the basic block
- Code Size Increase
  - Explicit NOPs may be numerous
  - Possible solution: code compression techniques
    - Used in modern VLIWs, but require added circuitry
- Huge number of registers needed
  - Thus, the complexity of register file to FU transport is increased
  - Possible solution: clustered VLIWs
Challenges of VLIW Technology

- Binary Incompatibility
  - Architectures with same ISA but different VLIW bundle size are incompatible
  - Also, same ISA and same VLIW bundle size, but different number and types of FUs and FU latencies → still incompatible!
  - There is no fully satisfactory solution to this issue
  - *Just In Time Compilation* is a possible, if costly, solution → attempted in Transmeta Crusoe
  - In most cases, VLIW are simply employed in embedded systems, where binary compatibility is less important
Early VLIW Architecture

- Multiflow Trace (1987)
  - Designed by Josh Fisher
  - Based on Fisher's ideas for scheduling algorithms
    - Trace scheduling
  - Has several direct descendants
- Cydrome Cydra 5 (1987)
  - The numeric processor for this departmental supercomputer was a VLIW
Modern VLIW Architectures

- Intel Itanium IA-64 EPIC → only major general purpose VLIW processor
- Transmeta Crusoe → designed for low-power general purpose processing (laptops), using binary to binary compilation for x86 compatibility
- STMicroelectronics ST200 (based on HP/STM Lx) → digital video processing
  - STxP70 can also be configured as a 2-way VLIW
- AMD Radeon R600 series → unified shaders using VLIW cores
  - Not used in later series aiming at more GPGPU loads
Conclusions

- VLIW Architectures employ statically scheduled multiple issue to:
  - Reduce hardware complexity
  - Decrease the CPI by exploiting ILP
- VLIW Architectures are more suitable for application specific processors for use in embedded systems
  - Digital video processing
  - Graphics
- VLIW architectures face significant limitations when dealing with legacy code, since poor binary portability
- VLIW architectures strongly depend on compiler quality
VLIW Architectures:
Some Examples

Optional Section
STMicroelectronics ST200

- ST200 is an embedded media processor based on Lx
  - Lx was developed jointly by STM and HP, based on Multiflow
- ST200 is a clustered VLIW machine
  - Up to 4 clusters, executing with a single program counter
  - Each cluster is a 4-way VLIW processor
    - Can execute 4 instructions, of which only 1 branch, 1 load/store, and 2 multiplications (no hardware division)
    - 64 register bank per cluster, R0 hardcoded at 0, R63 link register
    - Supports predication via select instruction
- NXP Trimedia -

- Note: NXP was formerly Philips Semiconductors
- Five Execution Units => Five operations per clock issued
- 15 Read and 5 Write Ports on register File
  - Need 15 read ports for 5 Execution Units because each operation requires two operands and a Guard operand.
  - Guard operand makes each operation conditional based upon value of LSB of the guard operand => Predicated Execution.
  - 128 Registers (r0, r1 always 0)
- Multiple operation sizes:
  - 2 bits for NOP, 26 bits, 34 bits, and 44 bits.
Philips TM 1000/Multimedia Processor

- Video In
- Audio In
- Audio Out
- i2C Interface
- VLIW CPU
  - 32K I$ 16K D$
- Main Memory Interface
- SDRAM
- VLD Coprocessor
  - Huffman decoder
  - Slice-at-a-time
  - MPEG-1 & 2
- Video Out
- Timers
- Synchronous Serial Interface
- Image Coprocessor
  - Down & up scaling
  - YUV → RGB
  - 50 Mpix/sec
- TM 1000
  - PCI Interface
    - PCI (32 bits, 33 MHz)

CCIR601/656
YUV 4:2:2
38 MHz (19 Mpix/sec)

Stereo digital audio
i2S DC=100 kHZ

2/4/6/8 ch. digital audio
i2S DC=100 kHZ

i2C bus to camera, etc.

V.34 or ISDN
Front End

Down & up scaling
YUV → RGB
50 Mpix/sec
VLIW vs. EPIC

- VLIW approach:
  - Fixed format of operation fields in the bundle.
  - Implicit parallelism among operations in the bundle.

- EPIC (Explicitly Parallel Instruction Computer) approach:
  - Flexibility in the instruction format
  - Compiler detects ILP and indicates when an instruction cannot be executed in parallel with its successors.
IA-64: instruction set architecture; EPIC is type
  • EPIC = 2nd generation VLIW?

Itanium™ is name of first implementation (2001)
  • Highly parallel and deeply pipelined hardware at 800MHz
    • 6-wide, 10-stage pipeline at 800Mhz on 0.18 µ process

128 64-bit integer registers + 128 82-bit floating point registers
  • Not separate register files per functional unit as in old VLIW

Hardware checks dependencies
  (interlocks => binary compatibility over time)

Predicated execution (select 1 out of 64 1-bit flags)
The integer registers are configured to help accelerate procedure calls using a register stack:

- mechanism similar to that developed in the Berkeley RISC-I processor and used in the SPARC architecture.
- Registers 0-31 are always accessible and addressed as 0-31
- Registers 32-128 are used as a register stack and each procedure is allocated a set of registers (from 0 to 96)
- The new register stack frame is created for a called procedure by renaming the registers in hardware;
  - a special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure

- 8 64-bit Branch registers used to hold branch destination addresses for indirect branches
- 64 1-bit predict registers
IA-64 Registers

- Both the integer and floating point registers support register rotation for registers 32-128.
- Register rotation is designed to ease the task of allocating of registers in software pipelined loops.
- When combined with predication, possible to avoid the need for unrolling and for separate prologue and epilogue code for a software pipelined loop.
  - makes the SW-pipelining usable for loops with smaller numbers of iterations, where the overheads would traditionally negate many of the advantages.
Instruction group: a sequence of consecutive instructions with no register data dependences

- All the instructions in a group could be executed in parallel, if sufficient hardware resources existed and if any dependences through memory were preserved
- An instruction group can be arbitrarily long, but the compiler must explicitly indicate the boundary between one instruction group and another by placing a stop between 2 instructions that belong to different groups

IA-64 instructions are encoded in bundles, which are 128 bits wide.

- Each bundle consists of a 5-bit template field and 3 instructions, each 41 bits in length

3 Instructions in 128 bit “groups”; field determines if instructions dependent or independent

- Smaller code size than old VLIW, larger than x86/RISC
- Groups can be linked to show independence > 3 instr
## 5 Types of Execution in Bundle

<table>
<thead>
<tr>
<th>Execution Unit Slot</th>
<th>Instruction Type</th>
<th>Instruction Description</th>
<th>Example Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-unit</td>
<td>A</td>
<td>Integer ALU</td>
<td>add, subtract, and, or, cmp</td>
</tr>
<tr>
<td>I</td>
<td>Non-ALU Int</td>
<td>shifts, bit tests, moves</td>
<td></td>
</tr>
<tr>
<td>M-unit</td>
<td>A</td>
<td>Integer ALU</td>
<td>add, subtract, and, or, cmp</td>
</tr>
<tr>
<td>M</td>
<td>Memory access</td>
<td></td>
<td>Loads, stores for int/FP regs</td>
</tr>
<tr>
<td>F-unit</td>
<td>F</td>
<td>Floating point</td>
<td>Floating point instructions</td>
</tr>
<tr>
<td>B-unit</td>
<td>B</td>
<td>Branches</td>
<td>Conditional branches, calls</td>
</tr>
<tr>
<td>L+X</td>
<td>L+X</td>
<td>Extended</td>
<td>Extended immediates, stops</td>
</tr>
</tbody>
</table>

- 5-bit template field within each bundle describes both the presence of any stops associated with the bundle and the execution unit type required by each instruction within the bundle.
Itanium™ Processor Silicon

Core Processor Die

4 x 1MB L3 cache
<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>800 MHz</td>
</tr>
<tr>
<td>Transistor Count</td>
<td>25.4M CPU; 295M L3</td>
</tr>
<tr>
<td>Process</td>
<td>0.18u CMOS, 6 metal layer</td>
</tr>
<tr>
<td>Package</td>
<td>Organic Land Grid Array</td>
</tr>
<tr>
<td>Machine Width</td>
<td>6 insts/clock (4 ALU/MM, 2 Ld/St, 2 FP, 3 Br)</td>
</tr>
<tr>
<td>Registers</td>
<td>14 ported 128 GR &amp; 128 FR; 64 Predicates</td>
</tr>
<tr>
<td>Speculation</td>
<td>32 entry ALAT, Exception Deferral</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>Multilevel 4-stage Prediction Hierarchy</td>
</tr>
<tr>
<td>FP Compute Bandwidth</td>
<td>3.2 GFlops (DP/EP); 6.4 GFlops (SP)</td>
</tr>
<tr>
<td>Memory -&gt; FP Bandwidth</td>
<td>4 DP (8 SP) operands/clock</td>
</tr>
<tr>
<td>Virtual Memory Support</td>
<td>64 entry ITLB, 32/96 2-level DTLB, VHPT</td>
</tr>
<tr>
<td>L2/L1 Cache</td>
<td>Dual ported 96K Unified &amp; 16KD; 16KI</td>
</tr>
<tr>
<td>L2/L1 Latency</td>
<td>6 / 2 clocks</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>4MB, 4-way s.a., BW of 12.8 GB/sec;</td>
</tr>
<tr>
<td>System Bus</td>
<td>2.1 GB/sec; 4-way Glueless MP</td>
</tr>
<tr>
<td></td>
<td>Scalable to large (512+ proc) systems</td>
</tr>
</tbody>
</table>
Itanium™ EPIC Design Maximizes SW-HW Synergy

Architecture Features programmed by compiler:

Branch Hints | Explicit Parallelism | Register Stack & Rotation | Data & Control Speculation | Memory Hints

Micro-architecture Features in hardware:

- **Instruction Cache & Branch Predictors**
- **Fetch**
- **Fast, Simple 6-Issue Issue**
- **128 GR & 128 FR, Register Remap & Stack Engine**
- **Control Bypasses & Dependencies**
- **Parallel Resources**
  - 4 Integer + 4 MMX Units
  - 2 FMACs (4 for SSE)
  - 2 L.D/ST units
  - 32 entry ALAT
- **Memory Subsystem**
  - Three levels of cache: L1, L2, L3

Speculation Deferral Management
10 Stage In-Order Core Pipeline

Front End
• Pre-fetch/Fetch of up to 6 instructions/cycle
• Hierarchy of branch predictors
• Decoupling buffer

Instruction Delivery
• Dispersal of up to 6 instructions on 9 ports
• Reg. remapping
• Reg. stack engine

Execution
• 4 single cycle ALUs, 2 ld/str
• Advanced load control
• Predicate delivery & branch
• Nat/Exception/Retirement

Operand Delivery
• Reg read + Bypasses
• Register scoreboard
• Predicated dependencies
Itanium processor 10-stage pipeline

- Front-end (stages IPG, Fetch, and Rotate): prefetches up to 32 bytes per clock (2 bundles) into a prefetch buffer, which can hold up to 8 bundles (24 instructions)
  - Branch prediction is done using a multilevel adaptive predictor like P6 microarchitecture

- Instruction delivery (stages EXP and REN): distributes up to 6 instructions to the 9 functional units
  - Implements registers renaming for both rotation and register stacking.
Itanium processor 10-stage pipeline

- **Operand delivery (WLD and REG):** accesses register file, performs register bypassing, accesses and updates a register scoreboard, and checks predicate dependences.
  - Scoreboard used to detect when individual instructions can proceed, so that a stall of 1 instruction in a bundle need not cause the entire bundle to stall

- **Execution (EXE, DET, and WRB):** executes instructions through ALUs and load/store units, detects exceptions and posts NaTs, retires instructions and performs write-back
  - Deferred exception handling for speculative instructions is supported by providing the equivalent of poison bits, called NaTs for Not a Thing, for the GPRs (which makes the GPRs effectively 65 bits wide), and NaT Val (Not a Thing Value) for FPRs (already 82 bits wide)
Itanium: 3 Levels of Cache

- L1 Cache: On-Chip Split I-Cache and D-Cache
  - I-Cache: 16 KB 4-way set associative, 32B block size
  - D-Cache: 16 KB 4 way set associative, 32B block size, dual port, write through, physically addressed and tagged

- L2 Cache: On-Chip Unified
  - 96 KB 6-way set associative, 64B block size

- L3 Cache: On-Package, 800 MHz
  - 4 MB, 4-way set associative 64B block size

- Translation Look Aside Buffer: 2 Levels
  - TLB: 32-entry L1 TLB + 96-entry L2 TLB
Transmeta Crusoe MPU

- 80x86 instruction set compatibility through a software system that translates from the x86 instruction set to VLIW instruction set implemented by Crusoe
- VLIW processor designed for the low-power marketplace
Crusoe processor: Basics

- VLIW with in-order execution
- 64 Integer registers
- 32 floating point registers
- Simple in-order, 6-stage integer pipeline: 2 fetch stages, 1 decode, 1 register read, 1 execution, and 1 register write-back
- 10-stage pipeline for floating point, which has 4 extra execute stages
- Instructions in 2 sizes: 64 bits (2 ops) and 128 bits (4 ops)
Crusoe processor: Operations

- 5 different types of operation slots:
  - **ALU operations**: typical RISC ALU operations
  - **Compute**: this slot may specify any integer ALU operation (2 integer ALUs), a floating point operation, or a multimedia operation
  - **Memory**: a load or store operation
  - **Branch**: a branch instruction
  - **Immediate**: a 32-bit immediate used by another operation in this instruction

- For 128-bit instr: 1st 3 are Memory, Compute, ALU; last field either Branch or Immediate
80x86 Compatibility

- Initially, and for lowest latency to start execution, the x86 code can be interpreted on an instruction by instruction basis.

- If a code segment is executed several times, translated into an equivalent Crusoe code sequence, and the translation is cached.
  - The unit of translation is at least a basic block, since we know that if any instruction is executed in the block, they will all be executed.
  - Translating an entire block both improves the translated code quality and reduces the translation overhead, since the translator need only be called once per basic block.

- Assumes 16MB of main memory for cache.